COSC 2306
Data Programming

Search

Binary search - recursive

```
def BinarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist) // 2
        if alist[midpoint] == item:
            return True
        elif item < alist[midpoint]:
            return BinarySearch(alist[:midpoint], item)
        else:
            return BinarySearch(alist[midpoint + 1:], item)

numbers = [1, 3, 5, 7, 9, 11, 13]
target = 7
result = BinarySearch(numbers, target)
print(f"Item {target} found? {result}")
```

Binary search - complexity

$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \ldots \rightarrow n/2^i \rightarrow \ldots$

i: number of steps

When search ends (size of search space is 1):

$n/2^i = 1$, so $i = \log n$

O(logn)

Practical consideration: sorting is an extra work

- For small n, sorting might not worth it, for very large n, sort can be expensive
- Sort once, search many times: the cost of sort is amortized and not significant

Binary search: pros and cons

- Pros:
  - Best average time cost among comparison-based search algorithms.
    - O(logn) is the lower bound.
- Cons:
  - Needs a sorted data structure
    - Pre-sorting of unordered list costs at least O(nlogn) average time, slower than O(n) sequential search.

Search

Can we do **O(1)** in a search?

What does **O(1)** really mean?

You can only make **1 (or a few)** comparison(s) to find the item

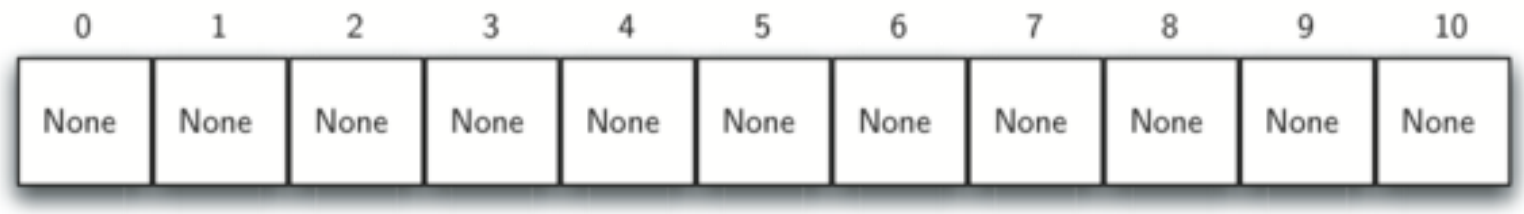
Hashing

- Key idea: Y = h(X), where X is the item and Y is the position, h is called **hashing function**

- This algorithm is called **hashing**, data is organized in a **hash table**

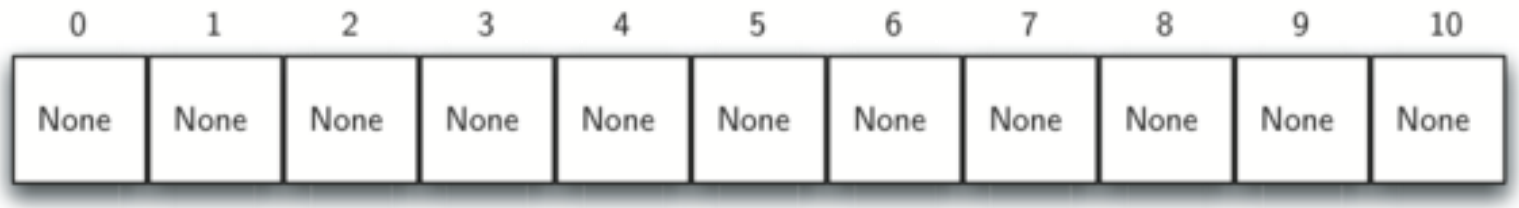
Hashing table

- **Slot** (bucket): each position of the hash table
- **Slot index** (name): an integer value (0, m-1)
- We can implement slot using a list
- We can initialize each slot as None
- Below is an example of Hash Table with 11 empty slots



Hash table

Q: how to map data into the slot?



A: use the **hash function h(X)** to determine where the data should be stored

Q: how to define a hash function?

Hash function

- The range of hash function h(X)?
  - If the length of hash table is **m**, **0 <= h(X) < m**

- Example: reminder method -- takes an item and divides it by the table length, returning the remainder (mode) as its hash value:

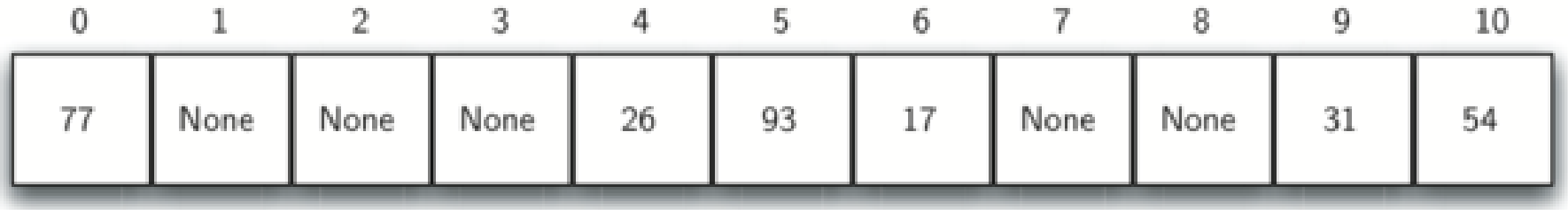
  h(item) = item%m

Assume m = 11

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Hash function

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9



Load Factor: λ = number_of_items/table_size

E.g., in the above example, λ = 6 / 11

Another example

I want to store the data of six students in a hash table of size 13

$h(k_1) = h(197354863) = 197354863 \% 13 = 4$	$h(k_4) = h(134152056) = 134152056 \% 13 = 12$
$h(k_2) = h(933185952) = 933185952 \% 13 = 10$	$h(k_5) = h(216500306) = 216500306 \% 13 = 9$
$h(k_3) = h(132489973) = 132489973 \% 13 = 5$	$h(k_6) = h(106500306) = 106500306 \% 13 = 3$

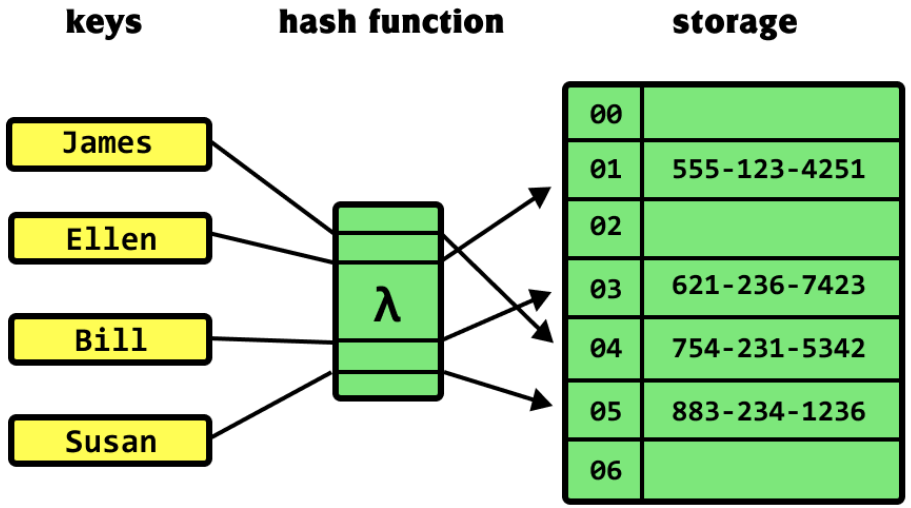
Suppose $HT[b] \leftarrow a$ means "store the data of the student with ID a into $HT[b]$." Then

$HT[4] \leftarrow 197354863$	$HT[5] \leftarrow 132489973$	$HT[9] \leftarrow 216500306$
$HT[10] \leftarrow 933185952$	$HT[12] \leftarrow 134152056$	$HT[3] \leftarrow 106500306$

Load Factor: $\lambda = 6 / 13$

HashTable coding

- Implement a map/dictionary style HashTable
  - Store key-value (or key-data) pairs
  - Key is used to look up the associated data value
  - The keys in a map are all unique -- a one-to-one relationship between a key and a data value
  - If a nonempty slot already contains the key, the old data value is replaced with the new data value



HashTable coding

- Coding logics
  - HashTable:
    - Initialize table size
    - 2 parallel lists: slot, data, both initiated with None
  - Hash function: reminder (key % table_length)
  - Insert item in HashTable (put)
    - Compute hash value using hash function
    - Put key and data directly to the slot
    - If key/data already exist, update the data
  - Search item in HashTable (get)
    - Use hash function to compute slot
    - Loop over HashTable
    - if slot == key, return

HashTable coding

- Coding logics
  - HashTable:
    - Initialize table size
    - 2 parallel lists: slot, data, both initiated with None
  - Hash function: reminder (key % table_length)

HashTable coding

- Coding logics
  - HashTable:
    - Initialize table size
    - 2 parallel lists: slot, data, both initiated with None
  - Hash function: reminder (key % table_length)

```
class HashTable:

    def __init__(self, hash_size):
        # Define a hash table with size of hash_size
        self.size = hash_size
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def hash_function(self, key, size):
        # Define a hash function using remainder
        return key % size
```

HashTable coding

- Coding logics
  - Insert item in HashTable (put)
    - Compute hash value using hash function
    - Put key and data directly to the slot
    - If key/data already exist, update the data

HashTable coding

- Coding logics
  - Insert item in HashTable (put)
    - Compute hash value using hash function
    - Put key and data directly to the slot
    - If key/data already exist, update the data

```
def put(self, key, data):
    hash_value = self.hash_function(key, len(self.slots))
    if self.slots[hash_value] is None:
        self.slots[hash_value] = key
        self.data[hash_value] = data
    elif self.slots[hash_value] == key:
        self.data[hash_value] = data  # replace data
    else:
        raise KeyError("Insert value failed!")
```

HashTable coding

- Coding logics
  - Search item in HashTable (get)
    - Use hash function to compute slot
    - Loop over HashTable
    - if slot == key, return